AFRL-RI-RS-TR-2011-239

COLLABORATIVE LEARNING FOR SECURITY AND REPAIR IN APPLICATION COMMUNITIES

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

OCTOBER 2011

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ AIR FORCE MATERIEL COMMAND ■UNITED STATES AIR FORCE ■ ROME, NY 13441

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2011-239 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
PATRICK M. HURLEY Work Unit Manager	WARREN H. DEBANY JR., Technical Advisor Information Exploitation and Operations Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* October 2011	**2. REPORT TYPE** Final Technical Report	**3. DATES COVERED** *(From - To)* July 2006 – April 2011

4. TITLE AND SUBTITLE

COLLABORATIVE LEARNING FOR SECURITY AND REPAIR IN APPLICATION COMMUNITIES

5a. CONTRACT NUMBER
FA8750-06-2-0189

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Michael Ernst
Martin Rinard
Jeff Perkins

5d. PROJECT NUMBER
AB41

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/Information Directorate
Rome Research Site/RIGA
525 Brooks Road
Rome NY 13441

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2011-239

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We investigated techniques that enable a system to learn where it is vulnerable to an attack or programming error, then automatically generate and evaluate ways that it can thwart the attack or recover from the error to continue to execute successfully. The approach is designed to work for systems, such as existing standard information technology installations, that have large monocultures of identical applications. By sharing information about attacks, errors, and response and recovery strategies, the system can quickly learn which strategies work best. The end result is a system whose robustness and resilience automatically grow over time as it learns how to best adapt and respond to the attacks and errors that its components inevitably encounter.

15. SUBJECT TERMS

Learning for security and repair, Attack detection and repair, Application Communities

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** PATRICK M. HURLEY
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	35	**19b. TELEPHONE NUMBER (*Include area code)*** N/A

Contents

List of Figures

1 Summary

We investigated techniques that enable a system to learn where it is vulnerable to an attack or programming error, then automatically generate and evaluate ways that it can thwart the attack or recover from the error to continue to execute successfully. The approach is designed to work for systems, such as existing standard information technology installations, that have large monocultures of identical applications. By sharing information about attacks, errors, and response and recovery strategies, the system can quickly learn which strategies work best. The end result is a system whose robustness and resilience automatically grow over time as it learns how to best adapt and respond to the attacks and errors that its components inevitably encounter.

We implemented two complementary approaches in the two phases of the project. In both phases, our technique first observes normal executions to learn the program's intended behavior. When errors are uncovered that enable attacks, patches are created that may restore learned values and eliminate the exploited vulnerability. Finally, it evaluates each patch, distributing the most successful one.

In both phases, we built a prototype system and applied it to Firefox 1.0 in the context of a Red Team exercise. In phase 1, the system recognized 100% of structural attacks and recovered from 83% of those attacks. No false positives were found. In phase 2, the system detected and repaired 92% of semantic attacks with an overhead of less than 2%. There were 2.1% false positives. We also performed other activities as part of this project.

2 Introduction

Most information technology installations today deploy many identical instantiations of the same software product on multiple machines. Because of this monoculture, a single error or security vulnerability can cause the entire system to fail catastrophically as all instantiations fail when presented with the same attack or error. Some especially important manifestations of attacks or errors are the presence of injected code, damage to the information representation of the program, and semantic errors where the program maintains its structure, but doesn't implement its intended specification

Ideally, it would be possible to use learning to change the weakness of this monoculture (many instantiations of the same application) into a strength. Specifically, one could learn from the failure of one instantiation to identify the error or vulnerability responsible for the failure, develop a mechanism to protect all instantiations from the worst effects of the error or attack and to enable the applications to recover to continue to execute after the error or attack. If there are multiple possible mechanisms or mechanism variants, it would be desirable to try the different variants to see how they work out. It would be then possible to identify the most effective variants and discard less effective variants.

To be effective, such an approach must work with standard COTS stripped windows binaries (such as Microsoft Office, Microsoft servers, Firefox, etc). Neither source nor debugging information is available for such software.

We proposed three approaches to automatically address security vulnerabilities in COTS software.

Targeted bounds checking is a technique that applies bounds checking in combination with continued execution to parts of the application that have been found to contain an error or vulnerability. This technique uses the presence of injected code to locate parts of the program that have a bounds error, then generates a variety of patches that use bounds checking to execute through the error. The goal is to use the availability of multiple instantiations of the same application to have the applications collaborate so that one instantiation can locate the vulnerability and pass the information along to other instantiations, which then try various alternatives to evaluate which patches work best.

Data structure consistency enforcement is a technique that leverages the presence of multiple instantiations of the same application to learn important data structure consistency constraints and the best ways to repair such constraints. This technique will combine observations from multiple instantiations to build a set of constraints that the program is expected to observe. These constraints will be distributed to the other instantiations, which will apply various data structure repair strategies (including no repair at all) and combine their results to determine which strategies work best.

Interface checking/enforcement is a technique that checks the operations of a program at critical interfaces such as operating system interfaces. The operating system interface is particularly important because that is where the program can affect the rest of the system. This technique learned context sensitive constraints over critical system interfaces (such as exec, open, etc) across multiple instantiations to build a set of constraints. These constraints are distributed and checked across the community. When they are violated repairs are created and evaluated across the community to determine which strategies work best.

In phase 1 of the project, we concentrated on structural errors that enable binary code injection attacks. These attacks inject arbitrary binary code into the application and execute it. This allows the attacker complete control over the users machine with all of the privileges of the user. Binary code injection attacks often utilize out-of-bounds accesses, but they also exploit a variety of other errors. These include dead pointers, garbage collection, unchecked arguments to script functions, and uninitialized memory.

In phase 2 of the project we concentrated on semantic errors that allow attackers to force the program to act in a manner contrary to its intended specification. These errors can enable simple actions like reading or writing an attacker controlled filename or actions that allow the attacker to execute arbitrary code (by controlling scripting code such as JavaScript or Virtual Basic).

We started the project with a solid foundation of existing techniques and implementations. Specifically, we had: the Determina Managed Program Execution Engine (MPEE), the Determina Memory Firewall injected code detector, the Determina client API for program instrumentation, the Daikon system for dynamically learning invariants, and the Determina LiveShield system for injecting patches into live applications. Each of these systems had worked well in their previous applications. One of the goals of the research was to build an integrated system based both on these components and some new components.

In phase 1, we used the Determina client API to build a value profiler for COTS windows software that gathered the values of registers and memory used by the application. Determina augmented the client interface to support this development and their system software to communicate the results of local Daikon analysis to a central server where the results could be aggregated. We augmented Daikon so that it could incrementally add invariant information as it was created to a central database. We also added a enhanced Daikon with additional invariants and support for basic block program points.

We built a plug-in to the Determina Management Console that received data about attacks and generated patches to check the behavior of learned invariants in the area of the attack. Once sufficient information about the attacks was received, it also generated and evaluated repairs. All of the patches were distributed and their results monitored using the Determina LiveShield system. We integrated all of these components together into a prototype Collaborative Learning for Security and Repair (CLSR) system.

Near the end of phase 1, Determina was acquired by VMware and VMware did not continue development on the products that we were using. We thus used some different available tools and developed some of our own during phase 2.

We built a profiler for COTS windows software that monitored critical system calls using the Pin dynamic instrumentation package. We captured both the arguments to the system calls and the dynamic call stack at the time of the call. This allowed us to build a model of behavior that was context sensitive.

We also built a centralized communications system that allowed us to gather behavior information for multiple clients and built a combined database. Our communications system also communicated the resulting constraints to each client as it started. When violations were found, the central system created patches and monitored their results across all of the clients.

We have detailed these and other changes in our quarterly reports.

3 Methods, Assumptions, Procedures

We adopted an experimental approach to our research. We chose Firefox 1.0 as a vehicle because of the relatively large number of documented security exploits. We reproduced and examined a number of these exploits to understand how the system might work and what information about the running program the system would need to know. In general, we observed the results we obtained and the general process of obtaining these results and used them to drive further development. We also identified any weaknesses or missing pieces and worked towards remedying the weaknesses and filling in any missing pieces. We also made parts of our software available for download via the Internet.

During the course of the project we devoted a major effort to integrating and evaluating the various different components. Our integration efforts focused on developing software to connect the different components. Once the software was developed we tested it and updated it as the tests indicated was necessary. We evaluated our techniques by applying them to different exploits. During this process we observed any deficiencies and developed techniques that addressed these deficiencies.

The underlying assumption behind this research is that these empirical techniques will generalize to larger classes of programs. We see no way to test these assumptions other than testing our techniques out on programs.

In both phase 1 and phase 2, in the later part of the project our activities centered around a Red Team exercise. The general idea behind a Red Team exercise is to set up an adversarial contest between a Red Team (in this case, a group from Sparta, Inc.) and a Blue Team (in this case, the MIT group). A White Team (from Mitre) officiated the contest.

In each phase, the Red Team exercise centered around a specific set of Firefox exploits. The Red, Blue, and White teams agreed that these exploits were representative of the entire set Firefox 1.0 binary code injection exploits in phase 1 and of the entire set of semantic exploits in phase 2. A subset of all of the available exploits was chosen to simplify the exercise and to limit the amount of time spent in replicating exploits

Prior to the exercise we used our learning tool to automatically learn key consistency constraints by observing its executions on test inputs that exercised the code related to the exploits.

4 Phase 1 - Structural Errors

Structural errors are those that violate the programs basic structure. They can be identified without reference to the program's specification because the violated rule (e.g., don't write off the end of a buffer) is not program specific.

Binary code injection attacks take advantage of structural errors to inject arbitrary code into the application and execute it. This allows the attacker complete control over the users machine with all of the privileges of the user. Binary code injection attacks often utilize out-of-bounds accesses, but they also exploit a variety of other errors. These include dead pointers, garbage collection, unchecked arguments to script functions, and uninitialized memory.

4.1 System Architecture

This section describes the high level architecture of CLSR. The architecture consists of the basic modules that make up the system and the communication between those modules. The architecture of the system is parameterizable: new modules can be substituted in, for instance to add new detectors or make other enhancements.

The system has two primary modes that are described separately, to ease the explanation. The two modes are normally used simultaneously in a community protected by the system.

The first mode is the learning mode. In this mode, the system learns the constraints that hold during normal operation.

The second mode is the protection mode. In this mode, the system monitors user applications for problems, logs the results of constraints related to the problem, and then creates and evaluates different repair strategies.

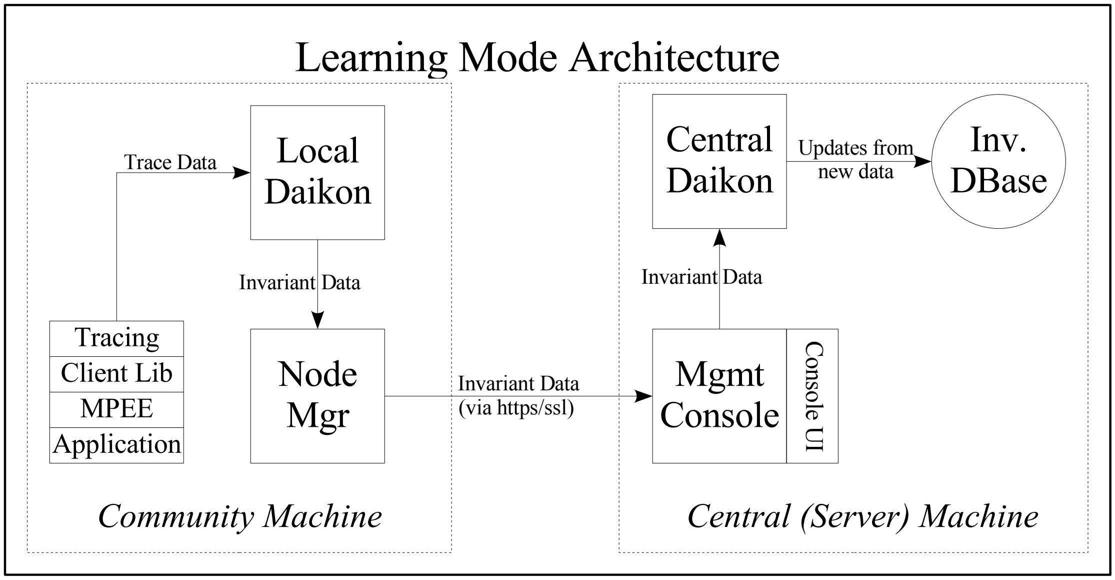


Figure 1: Learning mode architecture. The Daikon invariant detector learns locally valid constraints and merges the results from each member of the community centrally. The resulting constraints in the invariant database are valid over all traces collected on all machines. Only one community machine is shown, but thousands can be supported.

4.1.1 Learning Mode

Figure 3 (page 12) shows the learning mode architecture.

Constraints are learned over variables in each basic block within the target application. The application is instrumented with the MPEE. This engine allows machine code to be instrumented with minimal overhead. The MPEE provides a client library so that custom instrumentation can be written. The *Tracing* module uses the client library to instrument the application; the instrumentation sends the value of each variable to the local Daikon. Daikon uses machine learning to determine a set of valid constraints at each basic block.

The overhead of learning is high if the entire program is instrumented. The community allows this overhead to be amortized by instrumenting only a small percentage of the application on each machine. To reduce communication overhead, control of instrumentation is decentralized. Each machine instruments a random selection of the program. The community allows an extensive amount of data to be collected without burdening any particular machine.

Periodically, the locally valid constraints are sent to the central server. There the central Daikon merges those constraints with the constraints from all of the other community machines. The resulting *Invariant Database* contains constraints that are true across the entire community.

Network communication is built on HTTP/SSL standards, which facilitates communication across corporate LAN and WAN firewalls. All communication is authenticated on both sides. Inter-process communication (IPC) within a machine uses a variety of mechanisms. All IPC is encapsulated so that the implementation can be easily changed. Currently, these are not en-

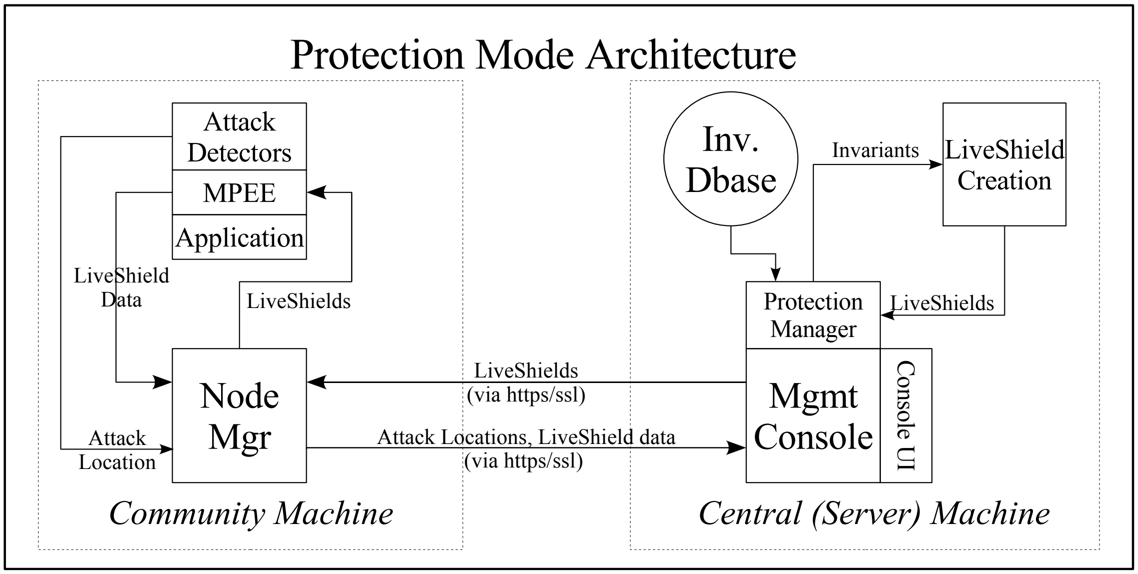


Figure 2: Protection mode architecture. Each community machine is in the monitoring, logging, or repair state. The Protection Manager (PM) coordinates system behavior during protection mode, including community machine transitions among the states. Attack detectors (such as Memory Firewall) inform the PM of attacks. The PM builds application patches (LiveShields) to determine which constraint violations are correlated with attacks and to implement possible repairs. The MPEE (Determina's Managed Program Execution Engine) applies those patches and returns results to the PM. Only one community machine is shown, but each community member has the same structure and connection to the central server.

crypted: this data reveals nothing more about security or privacy of the host system than is already available to anyone who has access to the host system.

4.1.2 Protection Mode

Figure 2 shows the protection mode architecture.

When responding to an attack, our system cycles through monitoring, logging, and repair states, under the control of the Protection Manager (PM). The PM which is loaded into the Management Console on the central server.

Initially, the system is in the *monitoring* state (see section 4.1.2). In the monitoring state, attack detectors (such as Memory Firewall) are used to detect attacks and their locations. When an attack is detected, the system enters the *logging* state (see section 4.1.2). Constraints related to the attack are logged and the logging data is used to determine which constraint violations are correlated with the attacks. In the *repair* state (see section 5.1.4), repairs are generated and distributed to the community. The effectiveness of different repairs is then analyzed, based on data logged by the repair LiveShields. Effective repairs are retained, and ineffective repairs are discarded.

The communication approach is the same as in the learning phase. Network communication is built on HTTP/SSL standards. Various mechanisms are used for IPC within a machine.

Monitoring An *attack detector* determines when an application has been attacked and reports the location of the attack. Two attack detectors are built into the system and others can be added.

Determina's Memory Firewall (which implements program shepherding [16])is used to detect code injection attacks. Memory Firewall examines each piece of code before it is executed the first time. Only code within the original application should be executed. Any other code must have been injected and its attempted execution indicates an attack.

Heap Guard is a new detector we developed that triggers when the application writes to the word immediately preceding or following an allocated heap buffer. Both Memory Firewall and Heap Guard have a negligible false positive rate.

When an attack occurs, the detector sends information about the attack to the PM, via the local Node Manager and the Management Console.

Logging When an attack is first detected, the system collects data to determine which constraints are violated when the attack occurs. The PM finds constraints in the invariant database that may be related to the attack. One way to find related constraints is to use the call stack and attack location returned by the attack detector. Constraints near those locations may be related.

Data about constraint violations during an attack is obtained by logging whether or not the constraint is violated each time the code is executed. This is accomplished by generating and applying a *LiveShield* for each related constraint. A LiveShield is a patch that can be dynamically applied to a program by the MPEE. A LiveShield patch consists of a conditional test and a payload (code to execute when the conditional test is true). When a LiveShield is executed it sends a message back to the PM with the state of the conditional (true or false).

LiveShields generated for logging include the conditional test but do not have a payload. They do not change the execution of the program but simply return information about the state of the constraint. The PM analyzes the results from all of the logging LiveShields to determine which constraints are correlated with the attack.

Repair The PM generates one or more possible repairs for each constraint that was correlated with the attack. Each repair consists of a LiveShield. The LiveShield's conditional test determines if the constraint is violated. The payload code repairs the constraint. There may be several possible repairs (LiveShields) for a constraint. For example, consider the constraint that only procedure `f1` is called from a particular `jsri`. Some possible repairs that could be tried when a different procedure is called are:

1. Skip the call
2. Call procedure `f1`
3. Return early from the calling function

The possible repairs are distributed (as LiveShield patches) throughout the community. Each time a repair is attempted, the results are sent back to the PM. If the repair was successful, it will be used more often. If the repair is unsuccessful, it may be discarded. In time, only successful repairs against the attack will be used.

4.2 Results and Discussion

We next present our results automatically generating repairs to binary code injection attacks in Firefox V1.0. These are the results from the red team exercise.

4.2.1 Rules of engagement

Several months before the Red Team exercise, the Red Team had access to all of the Blue Team's source code, tests, and documentation, including design documents, presentations to sponsors, and the Blue Team's own analyses of weaknesses of the system. Thereafter, the Blue Team periodically provided updates of this information to the Red Team. Furthermore, the Blue Team answered the Red Team's questions about the system's design and implementation. This is much more information than is available to a typical attacker. After this preparation, the Red Team exercise itself occurred at MIT on February 25–29, 2008.

The protected application was a stripped x86 binary of Firefox 1.0.0. This choice was motivated primarily to maximize the ability of the Red Team to develop meaningful attacks. At the time of Red Team exercise there was a substantial amount of publicly available vulnerability information for this application. The Red Team was able to leverage much of this information when creating new attacks or attack variants. Another advantage was the availability of source code, which simplified the Red Team's job of understanding the Firefox code during the attack creation process.

The exercise was designed to focus on the research aspects of the project. Therefore, the Red Team was only permitted to attack Firefox and the Blue Team infrastructure. For instance, for convenience the Red Team was given a login on the Blue Team computers, but was not permitted to install a rootkit to steal Blue Team passwords and corrupt the system.

All of the attacks manifested as arbitrary control flow transfer. This is the most dangerous variety of attack, and it is extremely widespread in practice. Such attacks are detected by our system's monitoring component, but the system is not limited to control flow transfer attacks, and it is easy to plug in a different attack detector without affecting the rest of of the system.

Prior to the Red Team exercise, the Blue Team generated the system's invariant database by running Firefox on a small collection of real web pages. The Blue Team also provisioned a small community consisting of five clients and a centralized server hosting the system, and made this community available to the Red Team. Later, the Red Team attacked this community.

4.2.2 Attacks

The Red Team aimed to create exploits that CLSR could not detect or could not repair, to create exploits that would cause more damage than good (such as enabling a different attack), to mislead CLSR by interleaving multiple variants of an attack ("polymorphic attacks") or completely different attacks, to attack the CLSR infrastructure, etc.

The Red Team attacked Firefox with exploits for 10 vulnerabilities. Each exploit took the form of a web page that caused Firefox to run arbitrary code of the attacker's choosing. The exploit was launched by navigating Firefox to the web-page's URL. All of these exploits targeted known vulnerabilities, although in some cases the attacks themselves were developed by the Red Team for the Red Team exercise. The targeted vulnerabilities included unchecked JavaScript types, out of bounds array accesses, uninitialized memory accesses, garbage collection problems, stack smashing, and heap overflows.

The Red Team deployed the attacks as follows:

Single exploit at a time: For each exploit, the Red Team repeatedly deployed the exploit against the community, varying the attacked machines during the deployments. This measured how many deployments (attack instances) CLSR had to observe before creating a patch that protected the entire community.

Variants: For 3 of the vulnerabilities, the Red Team developed several different exploit variants that all attempted to exploit the same vulnerability. The Red Team deployed each variant individually. The Red Team also tested whether a patch for one attack provided immunity against other variants.

Multiple exploits at a time: The Red Team deployed sets of exploits in an interleaved fashion. This determined whether CLSR could find and apply successful patches even when under attack from multiple simultaneous exploits, without CLSR's actions for different exploits interfering with one another.

After each patch was installed, the Red Team navigated Firefox to 53 real web pages to evaluate whether the repair patches negatively affected Firefox.

4.2.3 Results

Because the monitoring stage detected and prevented all attacks (by terminating the Firefox execution before the attack took effect), none of the attacks succeeded in subverting the protected version of Firefox. After observing an average of 8.3 failed attacks, CLSR generated successful patches for 7 of the 10 attacks. Those patches permitted the patched versions of Firefox to continue and to successfully display the evaluation pages after the attack. The Red Team compared the patched, attacked Firefox with standard Firefox (not under attack nor protected by our system). Qualitatively, users observed no negative effect. Quantitatively, there was also no effect; for example, rendered web-pages were pixel-for-pixel identical.

For several attacks, the Red Team developed several distinct variants. For each variant, our system found and applied the same successful patch as for the original attack. Furthermore, after exposure to one variant, the patch provided immunity against all other variants. This suggests that our system's patches address the root cause of the vulnerability.

When exposed to multiple attacks or multiple variants at a time, our system found and applied the same successful patches that it found when the Red Team deployed only one attack, and within the same number of exploit deployments for each attack.

The system suffered no false positives. The Red Team was unable to cause it to deploy a patch when no attack was underway — that is, when visiting a non-attack web page. The Red Team was also unable to subvert any patches to have any negative consequences other than terminating the application (which is exactly what would have happened anyway in the absence of the patch).

On the first day of the exercise, a bug in the system's filename handling caused it to re-use a file of old results. As permitted by the rules of engagement, the Blue Team fixed this minor error and the Red Team re-ran its experiments. On the last day of the exercise, the Determina communications infrastructure failed, but this was not the focus of the exercise and the Red and Blue Teams worked around the problem. The Blue Team did not make any other changes to the system during the exercise.

Exploits that were not patched The system was unable to generate a successful patch for 3 of the Red Team's exploits. The first failure occurred because during the Red Team exercise, the behavior comparison component was configured to only check invariants from the lowest procedure on the stack that had invariants. The relevant invariant appeared one procedure above this, so it was not even considered. Changing the configuration to include additional procedures on the stack would have enabled the system to generate a successful patch.

The second failure was because the learning suite did not provide sufficient coverage for Daikon to learn the appropriate invariant. We subsequently verified that, using an expanded learning suite, Daikon would have learned an invariant that would enable the system to generate a successful patch.

The third failure was because Daikon's invariants were not rich enough to capture the error. Daikon's less-than invariant relates two quantities, but in this case the appropriate property would have related the sum of a whole sequence of buffer lengths to another number. Learning such invariants would be possible, but substantially more expensive. Alternatively, Daikon could learn invariants over the arguments to `memcpy` and the last address in the buffer.

4.2.4 Performance and overhead

On average, when running on a single computer[1], CLSR took 5.5 minutes from the time of first exposure to a new attack to the time when it has obtained a successful patch for that attack. This is not the time required to stop a propagating attack — the monitoring stage prevents attacks from taking effect, so there is no propagation. Rather, these times reflect how long users must wait before they have a patched version of the program that provides continuous, uninterrupted service even while under attack.

The 5.5 minutes includes an average of 8.3 executions: to detect the attack and choose where to install invariant checking, to collect invariant checking results to determine correlated invariants, and to evaluate candidate patches. These numbers include one outlier, in which CLSR took 13 minutes to sequentially repair three distinct errors in the program, all of which could be exploited by the same attack. (After CLSR repaired one problem, the same exploit triggered an

[1] Dell 2950, two 2.3-GHz Xeon 4-core processors, 16Gbytes of RAM, Windows XP Service Pack 2, VMware VM under an ESX server.

error at a different place in the program which CLSR then detected and repaired, and so on.)

When CLSR is run on a community, the times would be shorter, since the invariant checking runs could be overlapped, as could the patch evaluation runs.

5 Phase 2 - Semantic Errors

Semantic errors are those that result in an implementation that does not match the intended program specification. For example, a browser should only allow files specified by the user to be uploaded to an external site. A coding error that allowed the external site to specify and upload arbitrary files is a semantic error.

CLSR2 focuses on protecting the confidentiality and/or integrity of the underlying operating system from semantic errors in the application. It learns normal behavior at the operating system call interface and detects deviations from normal behavior. CLSR2 can repair deviations. It monitors the results of possible repairs and chooses the repair which best maintains application behavior.

5.1 System Architecture

The basic system architecture of CLSR2 is similar to that of CLSR.

The system has two primary modes that are described separately, to ease the explanation. The two modes are normally used simultaneously in a community protected by the system.

The first mode is *learning mode*. In this mode, the system learns the constraints that hold during normal operation.

The second mode is *protection mode*. In this mode, the system monitors user applications for problems, logs the results of constraints related to the problem, and then creates and evaluates different repair strategies.

5.1.1 Learning Mode

Figure 3 (page 12) shows the learning mode architecture.

The PIN Dynamic Binary Instrumentation Tool[2] is used to apply instrumentation to designated system call sites, and possibly application-specific call sites, in the application under observation. For every system call encountered, we record elements of the *Context* of the system call. The context includes the functions on the stack at the time of the system call (the *stack trace*), function arguments (for functions whose signatures are known), and possibly additional information such as a trace of other recent calls, contents of registers, etc.

Daikon [11] uses machine learning to determine a set of valid constraints. The constraints found by Daikon depend on the grammar of properties, the variables over which the properties are checked, and the program points at which the properties are checked. Daikon checks each invariant in the grammar over each possible combination of variables at each program point. Daikon supports invariants with one, two, or three variables.

[2] http://www.pintool.org/

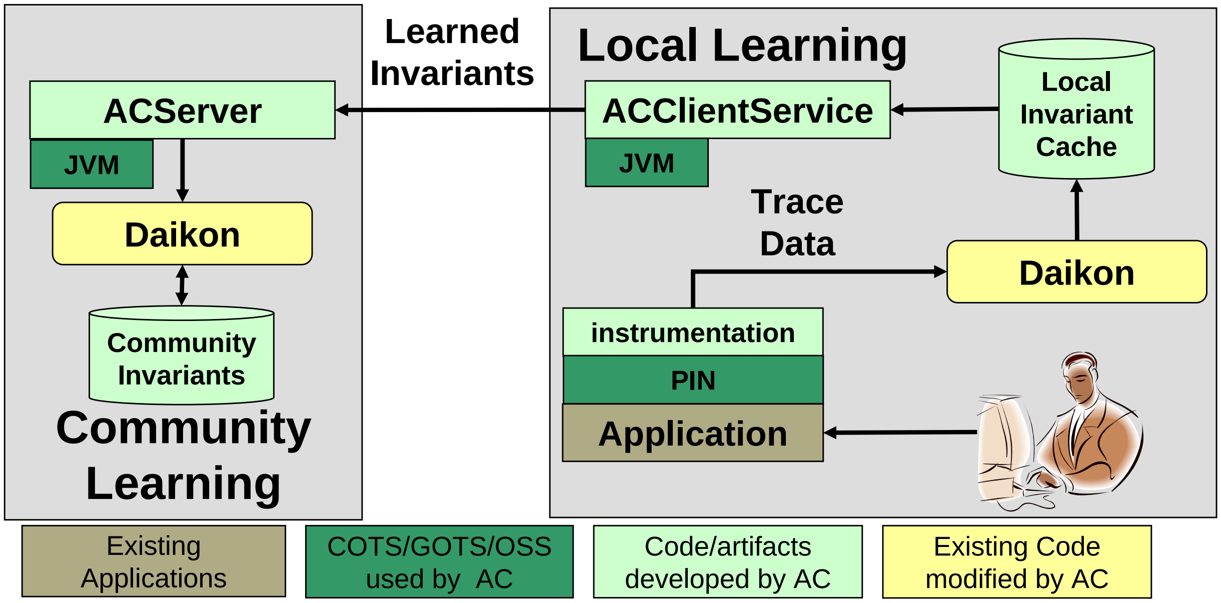


Figure 3: Learning mode architecture

Learning mode architecture. The Daikon invariant detector learns locally valid constraints and merges the results from each member of the community centrally. The resulting constraints in the invariant database are valid over all traces collected on all machines. Only one community machine is shown, but any number can be supported.

Program Points CLSR2 learns invariants at each possibly dangerous system call. As currently implemented on Windows these are `CreateFileW` (open), `DeleteFileW` (delete), and `ShellExecuteA` (exec). It is easy to add additional system calls. It is also possible to include application specific calls if it is desirable to apply the system to the internals of the application (and not just its interface with the operating system).

Each instance of a system call is considered to be a unique program point. Thus invariants learned at a `CreateFileW` in routine `foo` are separate from those learned from a call in routine `bar`. Furthermore, the analysis is context sensitive to the entire dynamic call chain. The precise location of each call (determined by its offset in its DLL) is used to identify each program point. See figure 4 for an example.

The dynamic call chain is modified to remove any duplicative entries. This ensures that two call chains that only differ by recursion depth will be treated as the same program point.

The context sensitive program points allow CLSR2 to learn very precise invariants which allows it to consistently identify attacks while suffering very few false positives.

Variables Each of the variables used in the system call are used as variables for invariant detection. For example, CreateFileW has 5 parameters (pathname, access, sharing, creation, and attributes).

```
01    int main() {            PPT 1
02      a(1); a(2);           main@2
03    }                       a@06
04                            b@11
05    void a (int n) {        c@15
06      if (n == 1) b();      DeleteFileW
07      else b()
08    }                       PPT 2
09                            main@2
10    void b() {              a@07
11      c();                  b@11
12    }                       c@15
13                            DeleteFileW
14    void c() {
15      DeleteFileW (``/tmp/test'');
16    }
```

Figure 4: Program point example
This C code results in two distinct program points (labeled PPT 1 and PPT 2). Each program point is identified by the list of active calls. Even though the same routines are called in each case, there are two different program points because routine b is called from two different call sites

CLSR also creates an additional variable (a *derived variable*) for each original parameter that is a pathname. The derived variable is the extension from the pathname. Derived variables are used for invariant detection in the same manner as are the original variables.

Grammar of invariants As implemented, CLSR2 relies on a few simple invariants:

- **Equality**

  Two variables are equal to one another ($x = y$).

- **OneOf**

  A variable is always one of a small list of constant values. For example, filename is always one of `/etc/startup` or `~/.start-up`. Or extension is always one of a set of image extensions (e.g., `jpg`, `gif`, etc).

- **Constant**

  A variable always has the same constant value. For example filename equals `C:/Windows/system32/shell32.dll`.

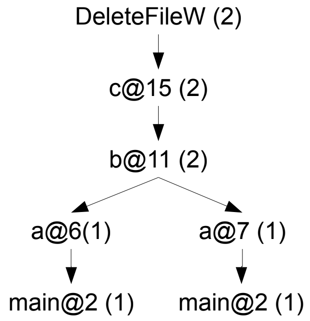


Figure 5: Combined program point tree
The combined tree for the program points identified in figure 4. The numbers to the right of each routine indicate the number of times that routine appeared in a call chain (at that level)

Call chain tree The learned stacks are combined into trees of related stacks for comparison. Each tree contains all of the stacks that were learned for the same system call and a subset of the arguments. For `CreateFileW`, the arguments are the extension, attributes, and access. For `DeleteFileW` and ShellExecA, the only argument used is the extension of the file name. For example, all of the stacks learned for `CreateFileW` with extension `jar` that were opened for `readonly` are combined together.

The tree is built by combining all of the related stacks. Each element in the tree is a routine from the call chain. The root of the tree is the system call. Each level of the tree has an entry for each routine at that level that calls the next lower level. For example, the combined tree for the call chains identified in figure 4 are shown in figure 5.

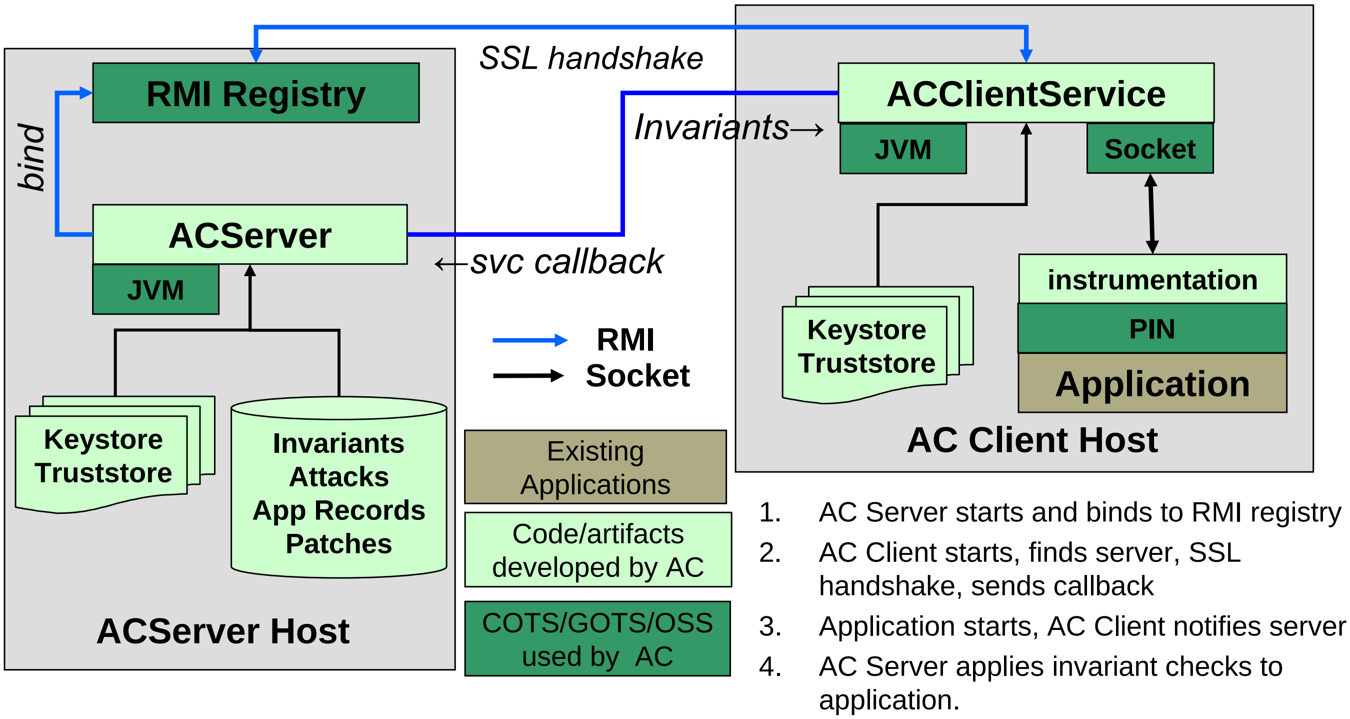
Distributed Learning The burden of learning constraints is distributed across the large number of workstations in the Application Community; that is, any individual workstation will only assume a small percentage of the overhead associated with learning.

The traces from all local executions are run through Daikon on the local workstation, which produces a set of constraints that are true for all runs on that workstation.

Periodically, the locally valid constraints are sent to the central server. There the central Daikon merges those constraints with the constraints from all of the other community machines. The resulting *Community Invariant Database* contains constraints that are true across the entire community.

5.1.2 System Start-up

Figure 6 (page 15) shows the critical events during system and application start-up. Prior to deploying the AC system, we will generate key and trust stores for the ACServer and the ACClientService instances. Each ACClientService will be configured to accept communications from monitored applications that are running on its local host only. These communications will utilize sockets and a simple ASCII message format. Communication between ACServer and ACClientService will use Secure Sockets Layer (SSL) / Transport Layer Security (TLS) rather than the default Java Remote Method Protocol (JRMP). ACServer binds to the RMI Registry, which is



1. AC Server starts and binds to RMI registry
2. AC Client starts, finds server, SSL handshake, sends callback
3. Application starts, AC Client notifies server
4. AC Server applies invariant checks to application.

Figure 6: Start-up architecture
Start-up architecture. First the AC Server starts up, and binds itself to the RMI Registry. When a workstation starts up, its client service sends a callback to the server (over SSL). When a protected application starts up, the client service notes the start-up and notifies the server.

running on the local host and is configured to only accept bindings to local applications. When an ACClientService starts on a remote node it will use its pre-positioned credentials to resolve the ACServer object that is published in the RMI Registry and to register itself with ACServer after an appropriate SSL handshake. The handshake will happen automatically because we will use javax.rmi.ssl.SslRMIClientSocketFactory and javax.rmi.ssl.SslRMIServerSocketFactory, both of which are included in the standard JDK version 1.6. When an instrumented application starts up it establishes a secure TCP connection to the local ACClientService and sends an initial ”start-up” message. The ACClientService responds by retrieving appropriate patches (invariant checkers, default repairs) from ACServer and passes them to the monitored application.

5.1.3 Protection Mode

Figure 7 (page 16) shows the architecture of *protection mode*.

There are two cases in which an attack is detected. It may be that a monitored call is detected in a context that does not correspond to any known program points recorded during the learning phase. If the call does correspond to a known program point, values from the current context are checked against the learned invariants for that program point.

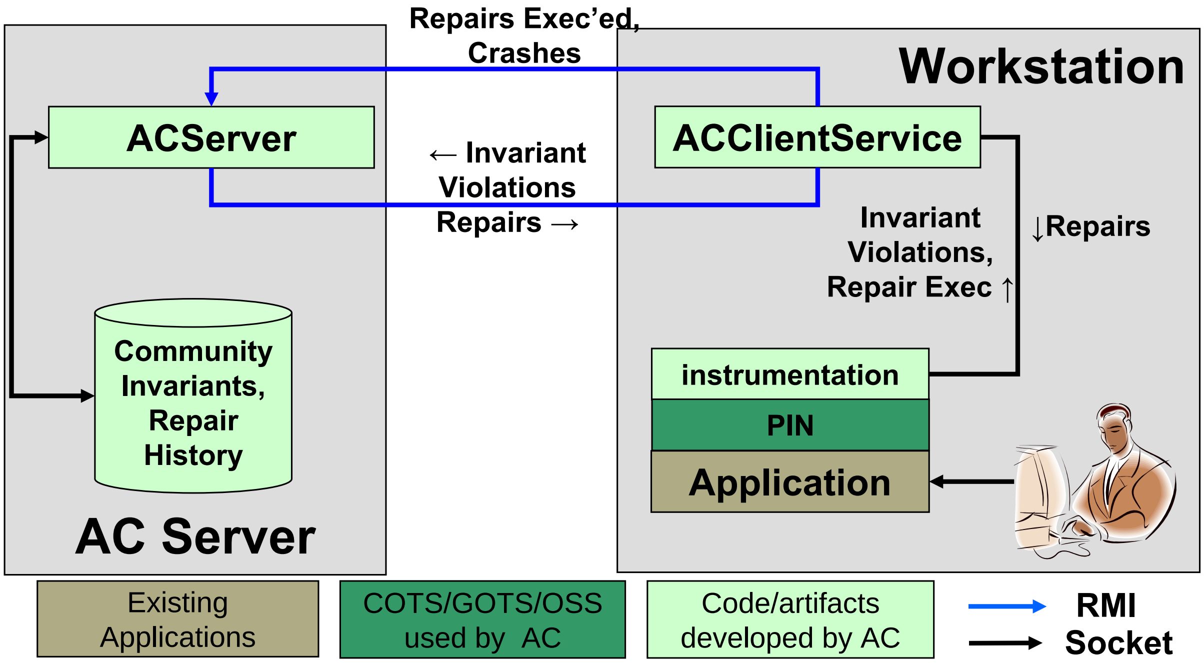


Figure 7: Protection mode architecture

Protection mode architecture. When an invariant violation is detected via instrumentation of an application, a repair is applied. When a repair is applied, a message to that effect is sent to the server. The server is also notified of any misbehavior of monitored applications. Only one community machine is shown, but each community member has the same structure and connection to the central server.

Comparing against a call chain tree When a previously unseen call chain is encountered it is compared against the call chain tree that matches its system call and arguments. The call chain is matched routine by routine starting at the system call until there is a mismatch. For example, if the call chain (`DeleteFileW`, `c@15`, `b@11`, `d@23`, `main@2`) were compared against the call chain tree in figure 5 it would match 3 levels deep (depth). At the point where the mismatch occurred, there were 2 branches (`a@6` and `a@7`) (branching).

The *depth* and *branching* for a call chain comparison are used to determine whether or not the call chain is considered to be normal behavior. CLSR2 implements two different checks depending on the system call and its parameters. CLSR2 considers some directories to be safe to read from. These are directories that are known not to contain files with sensitive information. For the most part these are directories that are either part of the standard Windows distribution or part of the standard application distribution. The default configuration of CLSR2 treats the Windows system32 directory (normally `C:/Windows/system32`), the Java installation directory (normally `C:/Program Files/Java`), and the application program directory (in this case `C:/ProgramFiles/Firefox`) as safe.

For reads from safe directories, CLSR2 considers any previously unseen stack with a depth of greater than 5 or a branching of greater than 3 to be normal operation. For any other operation, CLSR2 requires of depth greater than 15 or branching greater than 5 to consider the operation normal. Any operation this is not considered normal is treated as an attack and blocked.

A relatively large depth (e.g., 15) implies an operation that has been seen before albeit in a somewhat different context. A relatively large branching value indicates that this is a routine that is known to be called from a number of different places. This is what occurs when similar checks (such as lazy initialization) are placed throughout the code. A new source for this common call is unlikely to be an attack.

Invariant Violation Invariants are checked when a call chain matches a learned call chain exactly. In this case the arguments to the system call are checked against all of the constraints learned for the call chain. If any of the constraints are violated an attack has occurred.

5.1.4 Repair

If either type of violation occurs (unknown program point or invariant violation), the application applies a repair. In order to determine which repair strategy to apply, the application queries the server. The AC Server is responsible for tracking which repair strategies have been successful (or not) for which invariant violation.

After a repair is executed, a message to that effect is sent from application to client service and from there to the AC Server.

If "bad" behavior is detected (such as an application crash), a message is sent to the AC Server, and the Server may correlate misbehavior with repair execution and distribute different repair patches to application instances.

Figure 8 (page 18) shows a sequence diagram containing the interaction that occurs when an attack is detected.

The communication approach is the same as in the learning phase. Network communication is built on HTTP/SSL standards. Various mechanisms are used for IPC within a machine. The AC Server generates one or more possible repairs for each constraint that was identified with an attack. Some possible repair strategies when a system call is encountered that violates an invariant are:

- return an error code,
- enforce violated invariant,
- skip call,
- return immediately from calling method.

The possible repairs are distributed throughout the community. Each time a repair is executed, a message to that effect is sent to the AC Server. Because abnormal events (including crashes) are also communicated to the AC Server, the AC Server is able to consider which repairs appear to be the most successful with respect to maintaining nominal application behavior.

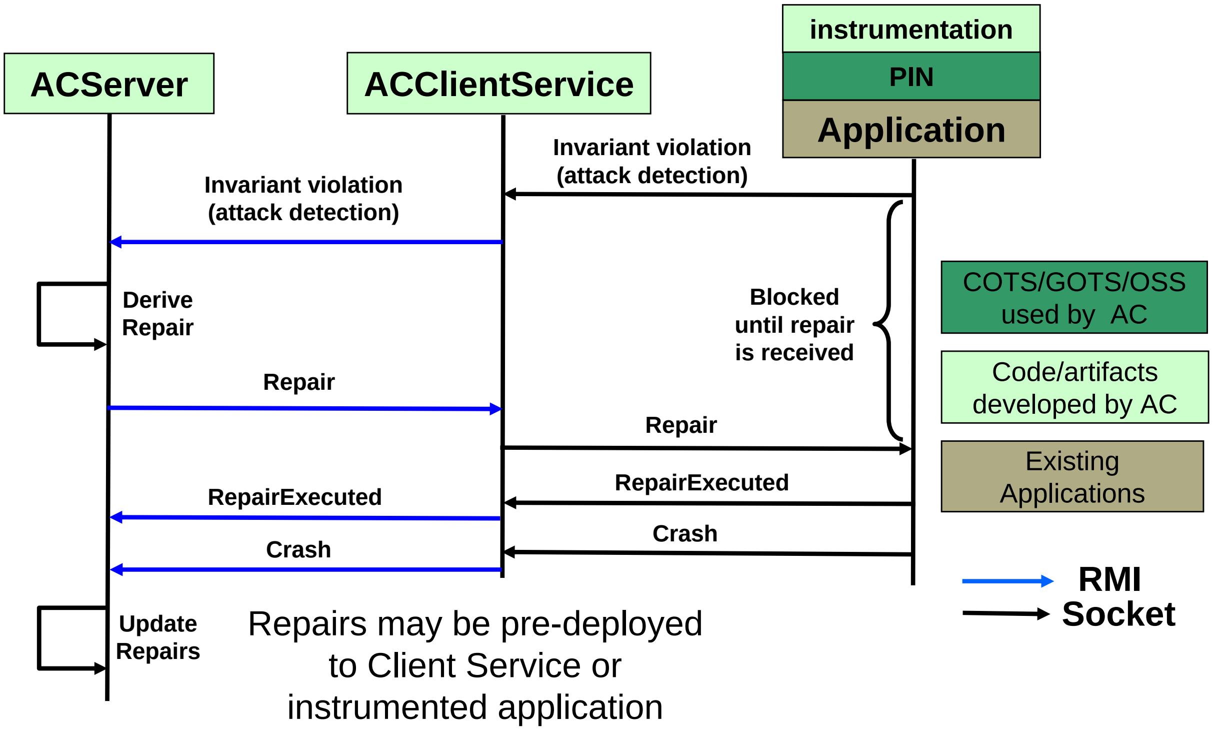

ACServer
ACClientService
instrumentation
PIN
Application
Invariant violation (attack detection)
Invariant violation (attack detection)
Blocked until repair is received
COTS/GOTS/OSS used by AC
Code/artifacts developed by AC
Existing Applications
Derive Repair
Repair
Repair
RepairExecuted
RepairExecuted
Crash
Crash
RMI
Socket
Update Repairs
Repairs may be pre-deployed to Client Service or instrumented application

Figure 8: Protection mode sequence diagram

Protection mode sequence diagram. Attack detectors (such as invariant checks) inform the AC Server of attacks. The AC Server builds application repair patches and installs them in the application instances. If the AC Server detects that a patched application crashes (or otherwise appears degraded), alternative repair patches are constructed and distributed. Only one community machine is shown, but each community member has the same structure and connection to the central server.

5.2 Results and Discussion

We next present our results automatically generating repairs to binary code injection attacks in Firefox V1.0. These are the results from the red team exercise.

5.2.1 Rules of engagement

Throughout phase 2, the Red Team had access to all of the Blue Team's source code, tests, and documentation, including design documents, presentations to sponsors, and the Blue Team's own analyses of weaknesses of the system. Furthermore, the Blue Team answered the Red Team's questions about the system's design and implementation. This is much more information than is available to a typical attacker. After this preparation, the Red Team exercise itself occurred at MIT on February 19–21 2010.

The protected application was a stripped x86 binary of Firefox 1.0.0. This choice was motivated primarily to maximize the ability of the Red Team to develop meaningful attacks. At the time of Red Team exercise there was a substantial amount of publicly available vulnerability information for this application. The Red Team was able to leverage much of this information when creating new attacks or attack variants. Another advantage was the availability of source code, which simplified the Red Team's job of understanding the Firefox code during the attack creation process.

The exercise was designed to focus on the research aspects of the project. Therefore, the Red Team was only permitted to attack Firefox and the Blue Team infrastructure. For instance, for convenience the Red Team was given a login on the Blue Team computers, but was not permitted to install a rootkit to steal Blue Team passwords and corrupt the system.

5.2.2 Learning for the Red Team Exercise

Because we did not have access to a sizable community, for the Red Team Exercise, we created two automated scripts for Firefox and also included some manual learning for areas that the scripts did not cover. The first script covered a set of user interface activities (such as using menus, typing into the location bar, etc). This was automated using a user interface recorder/playback mechanism. This script did not cover some things it should have due to difficulties in fully automating playback. The second script automatically loaded randomly chosen web pages.

Because timing changes the execution path of the application, we added random delays to the system calls so that we could cover more possible paths. We ran these scripts repeatedly to create the input files for the exercise. Both scripts were run thousands of times.

5.2.3 Attack Testing

The Red Team aimed to create exploits that CLSR could not detect or could not repair, to create exploits that would cause more damage than good (such as enabling a different attack), to mislead CLSR by interleaving multiple variants of an attack ("polymorphic attacks") or completely different attacks, to attack the CLSR infrastructure, etc.

The Red Team attacked Firefox with exploits for 50 vulnerabilities. Each exploit took the form of a web page that caused Firefox to execute system calls that violated the integrity and/or confidentiality of the underlying system. For example, reading/writing files or executing arbitrary programs. The exploit was launched by navigating Firefox to the web-page's URL. All of these exploits targeted known vulnerabilities, although in some cases the attacks themselves were developed by the Red Team for the Red Team exercise. The targeted vulnerabilities included a variety of JavaScript privilege escalation vulnerabilities, some spoofing attacks (such as saving images that were actually executables), and errors that allowed arbitrary files to be uploaded.

46 of the 50 tests were detected by CLSR and correctly repaired. In each of the 46 cases, Firefox was able to continue to operate correctly on subsequent inputs.

Four attacks were not detected. One worked only because it read a file in a directory (`C:\Windows`) that CLSR considered (via a system configuration) to be safe. This configuration exists because there are directories that only contain files that are available as a standard part of Windows and no information can be gained by reading those files. Note that when tested in a different directory CLSR blocked reading the file.

The other three attacks created a file but were unable to write to that file. The creation was allowed due to a bug in the check of the flags to the `open` system call. The checks is made because the system is more lenient of reads than it is of writes. The check incorrectly treated an open of read/write as read. When the check was fixed, each of the attacks was successfully detected and repaired.

5.2.4 False Positive Testing

The Red Team executed false positive tests to determine if CLSR's defensive steps prevent proper execution on benign web pages or reduce benign functionality of Firefox.

The false positive tests were run following completion of the system call attacks. The first phase of the false positive tests used 205 web pages. The pages were loaded from an Apache web server on the local MIT network. The second phase of the testing used a series of user interactions with the Firefox application. The set of user interactions are actions taken from the set of user interactions performed while executing the exploits but are benign in nature.

During the exercise, the Red Team recorded whether or not CLSR initiated a repair action on the web page and if CLSR interfered with the web page actions (file uploading, downloading, or other actions). The Red Team also captured the image of the browser as it rendered the web page. After the exercise, the Red Team compared the baseline values of the browser images and compared them to the images from the false positive testing. There were no differences in the web page appearance that could be attributed to CLSR. There were some trivial differences due to animated images or issues with the page capturing process.

There were 8 false positive detections related to user interaction, plus 3 instances of false detections on Delete File system calls that are believed to be associated with cache maintenance activities that had no noticeable impact on the use of the application. Of the 8 detections, 5 events counted as false positives. 2 events were on a single page and counted as 1 false positive. 2 events were detected by CLSR but did not cause an impact on the use of the application and were not counted against the false positive metric.

All of the false positives fell into one of three categories: file upload, file download, and cache file deletes. All of these occurred because of insufficient learning. File upload and download were mistakenly left out of the automated learning suites and there was insufficient manual testing to cover them thoroughly. The cache deletes occur only when the cache becomes full and that did not happen in the automated suites because Firefox was restarted from scratch after only a small number of web page loads.

When additional learning was added over these conditions, and the false positive tests were rerun, there were no false positive

5.2.5 Performance Testing

The Red Team executed two performance benchmarking tests to measure the performance impact of CLSR on the Firefox Browser. Each timed the loading of the 205 false positive test web pages with and without CLSRactive.

The first test used a local server and the second test used a remote server for the web pages. The local server showed 1.3% overhead. The remote server showed no measurable overhead.

6 Phase 3 - Commercialization

We investigated possible commercialization avenues for CLSR. We focused on Phase 2 because the system was simpler, required less learning, and had minimal overhead. We also had a dedicated management console for Phase 2.

6.1 Patent

In order to facilitate a licensing agreement with possible adopters, we filed a patent *Automatic Correction of Program Logic*. The abstract of the patent is as follows.

> An approach to detection and repair of application level semantic errors in deployed software includes inferring aspects of correct operation of a program. For instance, a suite of examples of operations that are known or assumed to be correct are used to infer correct operation. Further operation of the program can be compared to results found during correct operation and the logic of the program can be augmented to ensure that aspects of further examples of operation of the program are sufficiently similar to the examples in the correct suite. In some examples, the similarity is based on identifying invariants that are satisfied at certain points in the program execution, and augmenting (e.g., patching) the logic includes adding tests to confirm that the invariants are satisfied in the new examples. In some examples, the logic invokes an automatic or semi-automatic error handling procedure if the test is not satisfied. Augmenting the logic in this way may prevent malicious parties from exploiting the semantic errors, and may prevent failures in execution of the programs that may have been avoided.

6.2 Market Study

We commissioned a market study by Morgan Gregory (a student at the Sloan school with a background in software marketing) to better understand the market and who we should contact. The study found that our value position to possible adopters should be based around the ability of CLSR to detect and prevent zero day and target attacks at low overhead and with a low rate of false positives. It suggested contacting both large enterprise security companies and medium size companies that specialize in white-listing or blacklisting.

6.3 Competing Technologies

There are some products and companies with technologies that claim similar results to CLSR. These include ThreatFire (Symantec), Sana Security (AVG), and Prevx. These are all behavioral in approach, but we believe our context sensitive approach provides more fine grained detection (limiting false positives and false negatives).

7 Other Activities

In addition to the focus on the repair of security vulnerabilities, we performed a variety of other activities under this contract. We investigated various dynamic and static analysis approaches on both source and executable code as well as augmented source level type checking.

We also presented a demo of our techniques from phase 1 at DARPATECH 2007 (August 6 through August 9) in Anaheim California.

8 Conclusion

Security vulnerabilities pose an important threat to the integrity and utility of our computing infrastructure. Relying on manual programmer intervention to find and eliminate the errors that lead to vulnerabilities leaves systems open to exploitation for long periods of time. CLSR's automatic error detection and elimination techniques can provide, with no human intervention whatsoever, almost immediate protection against newly released attacks. The result is a program that is immune to the attack and can continue to provide uninterrupted service throughout and after the attack.

The Red Team evaluation showed that CLSR can automatically patch security vulnerabilities without introducing new attack vectors that the Red Team could exploit. CLSR does not solve every reliability and security problem, and there are many ways in which our technique and implementation can be enhanced. Nonetheless, this approach addresses an important and realistic problem, and it holds out the promise of substantially improving the integrity and availability of computing infrastructure.

9 Related Work

This research is a logical extension of previous work [9] in which we inferred data structure consistency specifications and created repairs [10] that enforce them. That work, too, was evaluated by a (different) hostile Red Team. By contrast to our current work, the previous approach fixed only data structures rather than entire applications, and handled a smaller class of execution problems; it worked on source code; it required a developer to review each inferred property for correctness; and it did no learning after invariant detection, deploying only patches that could statically be proved to terminate in a repaired state (with the effect that very few patches were ever deployed).

We discuss additional related work in attack detection mechanisms, continued execution in the face of attacks, checkpoint and replay techniques, and systems that combine attack detection and response.

9.1 Attack detection mechanisms

CLSR uses two attack detection techniques: program shepherding to detect and block malicious control flow transfers, and heap overflow checks to detect and block out of bounds writes to the heap. In general, however, CLSR can work with any attack detection technique that provides an attack location. StackGuard [6] and StackShield [31], for example, use a modified compiler to generate code to detect attacks that overwrite the return address on the stack. StackShield also performs range checks to detect overwritten function pointers. Researchers have also built compilers that insert bounds checks to detect memory addressing errors in C programs [2, 34, 4, 13, 24, 14, 15]. Dynamic taint analysis finds appearances of potentially malicious data in sensitive locations such as function pointers or return addresses [32, 7, 19]. It would be possible to make CLSR work with all of these detectors, although the high overhead and potential need for recompilation or even source code changes goes against CLSR's philosophy of operating on stripped Windows binaries and minimizing the overhead during normal execution.

9.2 Continued execution

CLSR's basic attack response mechanisms emphasize continued execution in the face of attacks. This philosophy is shared with failure-oblivious computing [23], acceptability-oriented computing [21], and boundless memory blocks [22], and in some cases the repair mechanisms are also similar. These techniques require no learning phase and no repeated executions for correlated invariant selection and evaluation. CLSR differs in deploying checks and repairs only to carefully targeted parts of the program and only in response to attacks, and in performing ongoing evaluation of each deployed repair. This makes CLSR's repairs more likely to be successful and less likely to disrupt the application. It also gives CLSR lower overhead even during the analysis of an attack, since the other techniques require bounds checks when applied to unsafe languages such as C.

Another related approach is transactional function termination [28, 26], which skips problematic computations much as CLSR can do. Transactional function termination responds to detected errors by undoing the effects of the function containing the detected error, then returning an error code to the caller [28, 26]. It relies on the caller's error processing code to make the continued execution successful. A difference is that transactional function termination is usually applied to skip and undo the effect of code that contains memory addressing errors that enable an attack, while the CLSR repairs skip the execution of the malicious code itself. The CLSR repairs can therefore eliminate vulnerabilities (such as the unchecked JavaScript type, uninitialized memory, and garbage collection exploits in Firefox) that are outside the scope of transactional function termination.

9.3 Checkpoint and replay

A traditional and widely used error recovery mechanism is to reboot the system, with operations replayed as necessary to correctly bring the system back up to date [12]. It may also be worthwhile to recursively restart larger and larger subsystems until the system successfully recovers [3]. Checkpointing [17] can improve the performance of the basic reboot process and help minimize the amount of lost state in the absence of replay. Checkpointing also makes it possible to discard the effects of attacks and errors to restore the system to a previously saved clean operational state. This technique, in some cases combined with replay of previously processed requests or operations that do not contain detected attacks, has been proposed as an attack response mechanism [29, 20, 27, 33].

In comparison with CLSR's continued execution philosophy, checkpointing plus replay has several drawbacks. These include service interruptions as the system recovers from an attack (we note that these service interruptions can occur repeatedly unless the system is otherwise protected against repeated attacks), the potential for replay to fail because of problematic interactions with external processes or machines that are outside the scope of the checkpoint and replay mechanism, lost state if the system chooses to forgo replay, and the extra system complexity associated with deploying the checkpoint and replay mechanism.

Dropping requests or operations that may contain an attack makes any information in the requests or operations unavailable to the user. It is entirely possible, for example, for useful information sources such as legitimate web pages, images, or presentations to become surreptitiously infiltrated with attack data. By enabling programs to execute successfully through such attacks, CLSR can make it possible for users to access the useful information despite the presence of the attack. The CLSR repair for one of the heap overflow errors in the Red Team exercise, for example, makes it possible for users to view useful image files that contain attacks (or even innocent data that happens to exercise the vulnerability).

9.4 Attack detection and response

The large overhead of many proposed protection techniques has inspired attempts to reduce the performance impact on programs in production use. One commonly proposed technique is to run heavily instrumented versions of potentially vulnerable programs on honeypots [30, 1], leaving the production versions unprotected against new attacks. When the honeypot is attacked, the instrumentation can detect the attack and develop a response that protects the production versions of the programs. This technique can be applied to virtually any attack detection and response mechanism that is too expensive to deploy directly to programs running in production.

DYBOC [26] uses honeypots that instrument buffer accesses to identify out of bounds accesses. The distributed attack responses identify which buffers to instrument (instrumenting all buffers is too expensive). Programs respond to attacks by using transactional function termination to avoid the out of bounds writes required for the attack to succeed. Vigilante [5] uses honeypots to detect attacks and dynamically generate filters that check for inputs that follow the same control-flow path as the attack to exploit the same vulnerability. ShieldGen [8] uses Vigilante's attack detection techniques to obtain attack inputs. It generates variants of the attack input and tests the variants to see if they also exercise the vulnerability. It then produces a general filter that detects all such variants. The filters can then be distributed to production machines to filter out attack inputs before they reach vulnerable programs.

It is also possible to deploy expensive attack detection and response mechanisms in a piecemeal fashion across a community of machines, with each program instrumenting only a small portion of its execution [18]. Machines that detect attacks can then distribute information that enables all machines to apply the technique to the appropriate region of the program and thereby survive the attack.

Sweeper [33] uses address randomization for efficient attack detection. This technique is efficient enough to be deployed on production versions of programs, but provides only probabilistic protection and therefore leaves programs still vulnerable to exploitation [25]. Sweeper uses attack replay in combination with more expensive attack analysis techniques such as memory access checks, dynamic taint analysis, and dynamic backward slicing. The generated filters discard attack inputs before they reach vulnerable programs. Sweeper also builds vulnerability-specific execution filters, which instrument selected instructions involved in the attack to detect the attack. The attack response is to use rollback plus replay to recover from the attack.

In contrast to many previous systems, CLSR employs an attack detection mechanism (Memory Firewall) that not only efficiently detects attacks, but also prevents attacks from taking effect at all. When CLSR succeeds in eliminating the vulnerability, the program can successfully process even attack inputs. In this case there is no need to deploy filters that discard attack inputs before they can reach the program. Also in contrast to many previous systems, which typically apply their attack response analyses across broad ranges of the program, CLSR uses the attack location to dramatically narrow down the region of the program that it instruments during its attack analysis and response generation activities. This makes it possible to deploy sophisticated but expensive analyses within this focused region of the program while still keeping the total overhead small. A potential downside of this focus is that CLSR may miss the invariant required to correct the underlying error in the program logic.

10 Symbols, Abbreviations, and Acronyms

AC: Application Communities

CLSR: Collaborative Learning for Security and Repair. The system developed as part of this project.

CLSR2: The version of the system developed for phase 2.

COTS: Commercial off-the-shelf. Normally refers to packaged software that can be purchased such as Microsoft Office, Firefox, etc.

CTO: Chief Technical Officer.

GOTS: Government off-the-shelf. Software Government products that are ready to use.

HTTP: HyperText Transfer Protocol. A networking protocol for distributive collaborative hypermedia information systems.

IPC: Inter-Process Communication.

LAN: Local Area Network.

MPEE: Determina Managed Program Execution Engine. This is the tool that supports dynamic binary instrumentation.

OSS: Open-source software. Computer software that is available in source code form.

PM: Protection Manager.

RMI: Remote Method Invocation. A Java mechanism for communicating between processes.

SSL: Secure Socket Layer. A communication protocal that uses encryption to protect transfers.

TLS: Transport Layer Security. This is part of SSL.

WAN: Wide Area Network.

11 Glossary

AC Server: Application Community server in CSLR2.

API Application Programming Interface. A set of function calls that provides a specifc piece of functionality.

Application Community: A set of computer systems that runs similar code for similar purposes.

Attack Detector: Code that determines when an application has been attacked and reports the location of the attack.

Daikon: A dynamic invariant detector.

Heap Guard An attack detector that triggers when the application writes immediately before or after a heap buffer.

Invariant Database: The set of invariants found to be true across all invocations in the community.

Learning Mode: The mode of the system where it learns invariants.

LiveShield: The Determina system for loading patches into running programs (based on the the MPEE).

Memory Firewall: A plugin to the MPEE that checks for injected code at indirect transfers of control.

Pin: A dynamic binary instrumentation tool similar to the Determina MPEE.

Protection Manager: CLSR Subsystem responsible for coordinating the response to attacks.

Protection Mode: The mode of the system where applications are protected from attack.

Semantic Errors: Errors that violate the program's *intended* specification (but not its underlying structure)

Structural Errors: Errors that violate the program's basic structure (e.g., overflows of stack frames or heap buffers)

12 References

[1] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *USENIX Security*, Aug. 2005.

[2] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, June 2004.

[3] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS-VIII*, pages 110–115, May 2001.

[4] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI*, June 2003.

[5] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, Oct. 2005.

[6] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security*, January 1998.

[7] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO*, Dec. 2004.

[8] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *IEEE S&P*, May 2007.

[9] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. 2006.

[10] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. 2005.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3), Dec. 2007.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX*, June 2002.

[14] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, May 1997.

[15] S. C. Kendall. Bcc: run-time checking for C programs. In *USENIX Summer*, 1983.

[16] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security*, Aug. 2002.

[17] M. Litzkow and M. Solomon. The evolution of condor checkpointing. In *Mobility: processes, computers, and agents*. ACM Press/Addison-Wesley, 1999.

[18] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *SNDSS*, Feb. 2005.

[19] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, Feb. 2005.

[20] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev.*, 39(5):235–248, 2005.

[21] M. Rinard. Acceptability-oriented computing. In *OOPSLA Companion*, Oct. 2003.

[22] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC*, Dec. 2004.

[23] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, December 2004.

[24] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *NDSS*, February 2004.

[25] H. Shacham, M. Page, B. Pfaff, E.-H. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *ACM CCS*, Oct. 2004.

[26] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *ISC*, Sept. 2005.

[27] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *IEEE S&P*, May 2007.

[28] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX*, Apr. 2005.

[29] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In *NDSS*, Feb. 2005.

[30] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2002.

[31] Stackshield. `www.angelfire.com/sk/stackshield`.

[32] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS*, Oct. 2004.

[33] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *EuroSys*, Mar. 2007.

[34] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *ESEC/FSE*, 2003.